Bias- Variance tradeoff

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	high variance	high bias	high bias & variance	low bias & variance

if train set error is large → high bias

if diff. between train set error & dev set error is large → high variance

Above all is true, if Optimal error / Bayes error (Human error) is ≈ 0% & train set & dev set is drawn from & the same distribution

Teacher's Signature

Basic recipe for ML

Try these stuffs

High bias ⟶ Use bigger network
⟶ Train longer data
(Try new NN architecture)

High variance ⟶ More data
⟶ Regularization
(Try new NN architecture)

bias → error your model has
It fails while capturing some
the features in data

Variance → overfits the data

Teacher's Signature

Regularization

Frobenius norm formula

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(w_{i,j}^{[l]}\right)^2$$

i of the matrix $\longrightarrow$ no. of neurons of l^{th} layer

j of the matrix $\longrightarrow$ no. of neurons in the previous layer $n^{[l-1]}$

[Hence, summation from $j=1$ to $n^{[l-1]}$]

$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} \cdot w^{[l]}$$

↳ new formula for gradient

$$w^{[l]} := w^{[l]} - \alpha\, dw$$

$$\therefore w^{[l]} = w^{[l]} - \alpha \left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

↑ prev. value of w

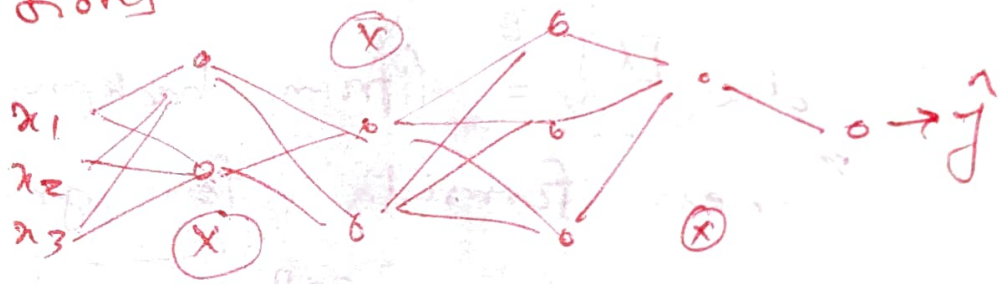
$$\left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]} = w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\,(\text{from backprop})$$

Dropout Regularization

Inverted dropout (Most popular version of dropout)

Turning some of the neurons in NN to zero (dropping them off) OR deactivating them.

We can do this by simply turning value of $w_{i,j}^{[l]}$ to zero rather than changing $w^{[l]}$'s dimensions



Say we want to reduce number of neurons in layer 3 by 20%, ie keep 80% of the neurons alive

```
d3 = np.random.randn(a3.shape[0],
                     a3.shape[1])
< keep-prob
```

keep_prob = 0.8

~~a^3 = np.multiply (a^3, d^3~~

a3 = np.multiply (a3, d3) } multiplying by d3

OR a3 *= d3

~~d3 is a matrix of 0's & 1's~~

d3 is a matrix of 0's & 1's

a3 /= keep_prob

Why divide a3 by keep_prob?

When you will multiply a3 with d3, some of the neurons will be set to zero. To compensate this, we are stepping up the remaining neurons by multiplying it with keep_prob. This done to preserve the expected value of $z^{[4]}$

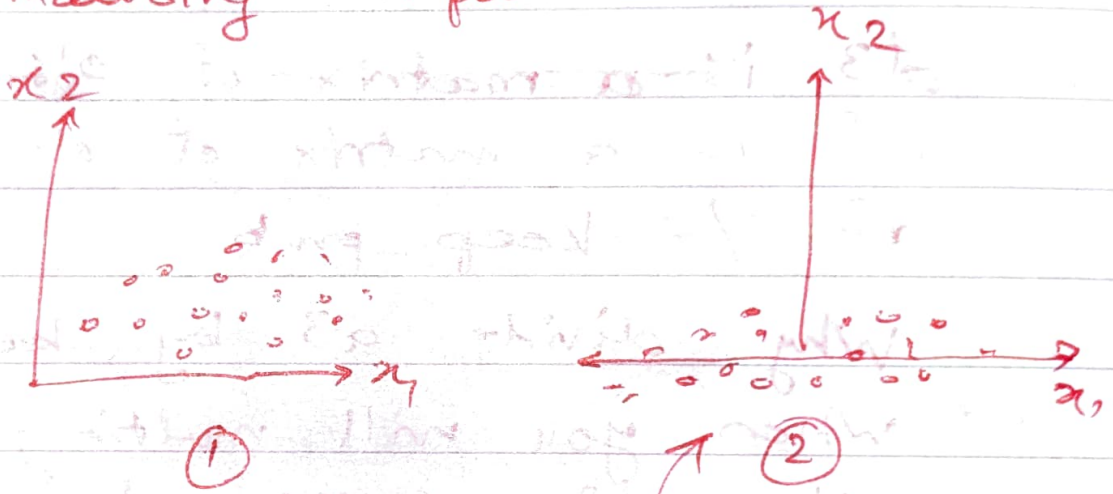
$$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$$

Hence, this is called inverted dropout

Teacher's Signature

do this only for training set not test set. You can tune value of keep_prob

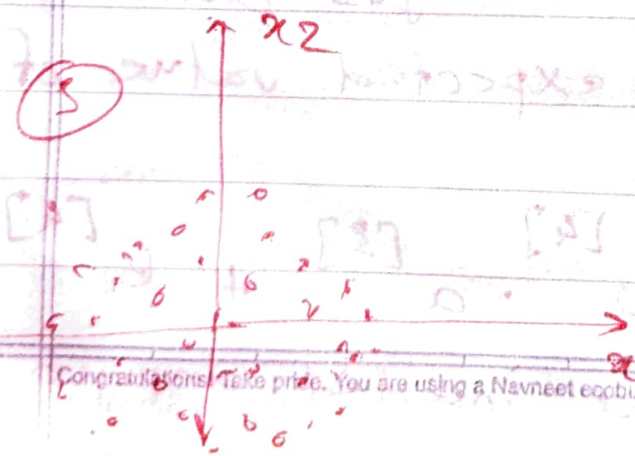
Normalizing input



Subtract mean

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$



x_1 has high variance than x_2. We can normalize this variance by performing

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} **2$$

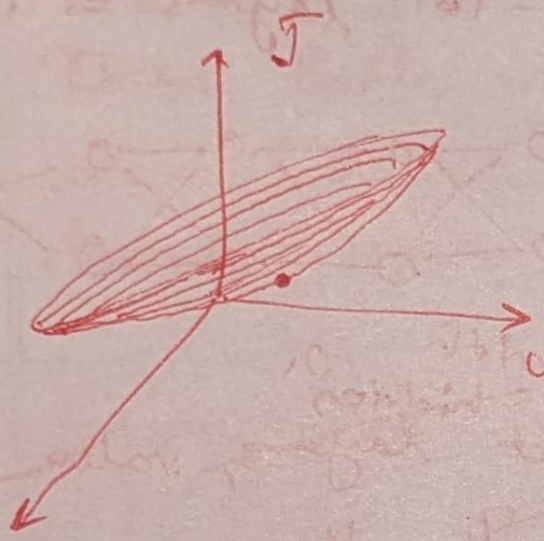
↑ Element wise

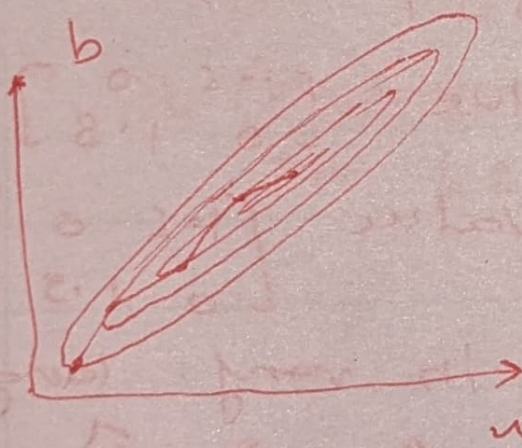
Teacher's Signature

$x := x / 6$

Why normalize inputs?

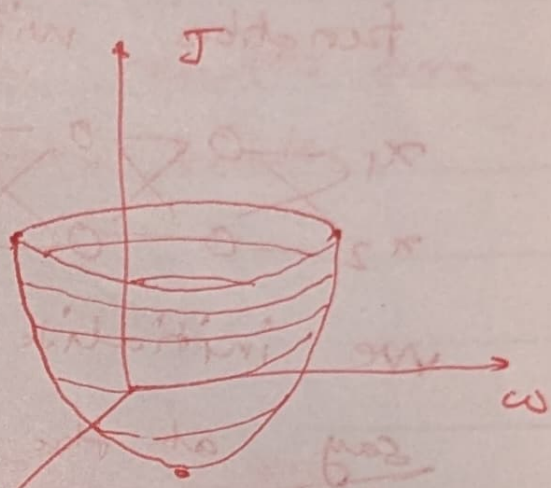
Unnormalised

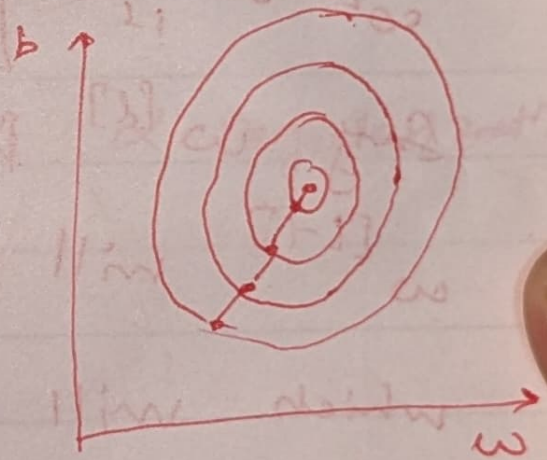




taking large value of α might affect learning time as gradient descent will take a lot of steps to oscillate back & forth to find

Normalized





kind of any value of ~~gradi~~ α as you can start from anywhere.

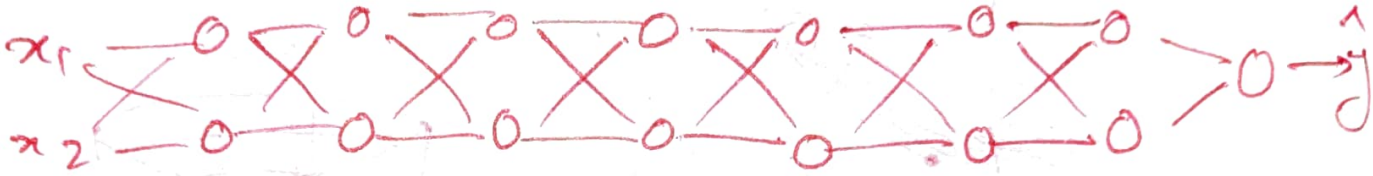
min.

Vanishing / exploding gradients

Caused due to initialization of w.

In a NN where there are many layers (L), this problem may arise.

say we have 2 features x_1 & x_2 and all the hidden also has 2 activation functions with 8-10 layers of it



we initialize b with 0.

say at the ~~output~~ last ~~hidden~~ layer, value of $z^{[L-1]}$ is $\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$, then

say $w^{[L]}$ has value $\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ for $w^{[L-1]}$ will have value $\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1}$

which will result in very large value of $w^{[L-1]}$. ~~And~~ $\hat{y}$.

$\therefore$ Gradient will explode.

Or if w has value $\begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix}$

or something less 1, gradient descent will vanish.

~~Thus~~ Both of these results are undesirable bcoz these make learning slow

keep_prob $\propto \frac{1}{\text{regularization}}$

λ (hyperparameter $\propto$ regularization)
↑ in L2 reg.)

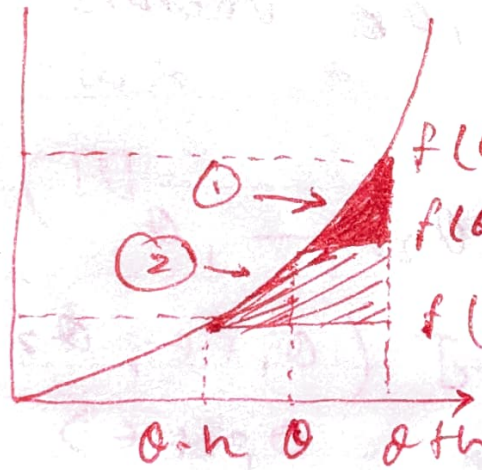
Teacher's Signature

Numerical approximation of Gradients

We know $\frac{d f(x)}{dx} = \lim_{x \to 0} \frac{f(x+h) - f(x)}{h}$

But, using $\lim_{x \to 0} \frac{f(x+h) - f(x-h)}{2h}$

is more accurate.



Say $f(\theta)$ is the function $= \theta^3$

previously we used only triangle ①

But we can use ① & ② to get more accurate results

$\therefore f'(\theta) = \lim_{h \to 0} \frac{f(\theta+h) - f(\theta-h)}{2h}$

If we want to check the computed gradients i.e. dw, db, da, dz, we can use the approximation technique.

We know J is function of

$(W^1, b^1), (W^2, b^2), \cdots (W^{[l]}, b^{[l]})$

OR $\theta_1, \theta_2, \cdots \theta^{[l]}$

what we can do is merge all the $\theta_1, \theta_2, \cdots, \theta^{[l]}$ vectors into single vector θ

$\therefore J(\theta) = J(\theta_1, \theta_2, \cdots, \theta^{[l]})$

$$d\theta_{approx.}[i] = \frac{J(\theta_1 + \epsilon, \theta_2 + \epsilon, \cdots \theta_i + \epsilon) - J(\theta_1 - \epsilon, \theta_2 - \epsilon, \cdots \theta_i - \epsilon)}{2\epsilon}$$

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

PAGE NO. :

DATE : / /

$d\theta_{approx} \cdot shape = d\theta_{approx} \cdot shape$

$= \theta \cdot shape$

check

L2 norm $\rightarrow \dfrac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$

Usually, $\epsilon = 10^{-}$

if the value of this is

close to 10^{-7} $\longrightarrow$ good res.

10^{-5}

10^{-3} $\longrightarrow$ worry

Teacher's Signature

PAGE NO. :

DATE : / /

Weight initialization

Instead of randomly generating weights, we can initialize them to a particular value for better performance

$$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$$

$n^{[l-1]}$ number of neurons in last layer

For ReLU, it is $np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

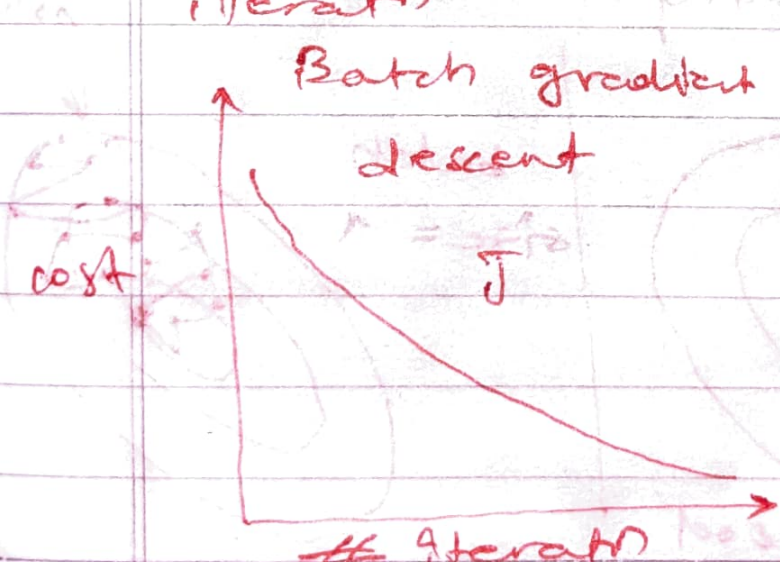
For tanh, it is $np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

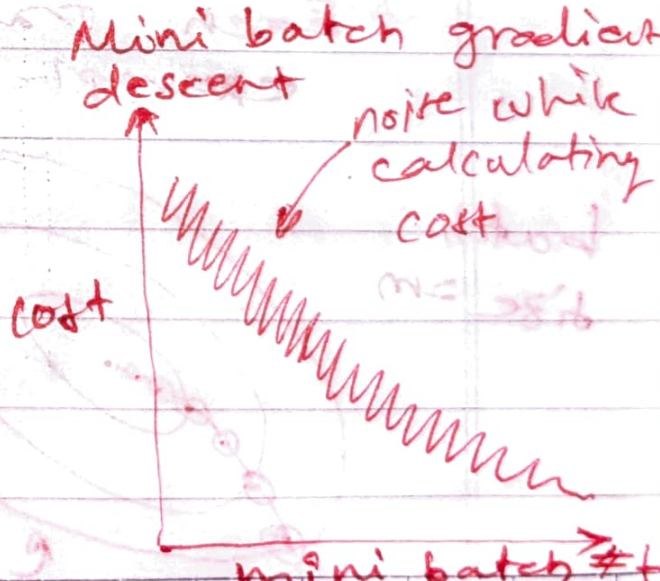
Teacher's Signature

Mini batch gradient descent

Instead of training on entire training set, we'll divide it into 't' parts. Then train model on each t part.

Say we divide into 1000 examples each. We'll perform forward prop on first 1000 examples, then perform back prop on those 1000 examples by calculating cost & g.d. for number of iterations





Plot $J^{\{t\}}$ using $X^{\{t\}}, Y^{\{t\}}$

Mini batch gd is ~~fast~~ & computationally cheaper

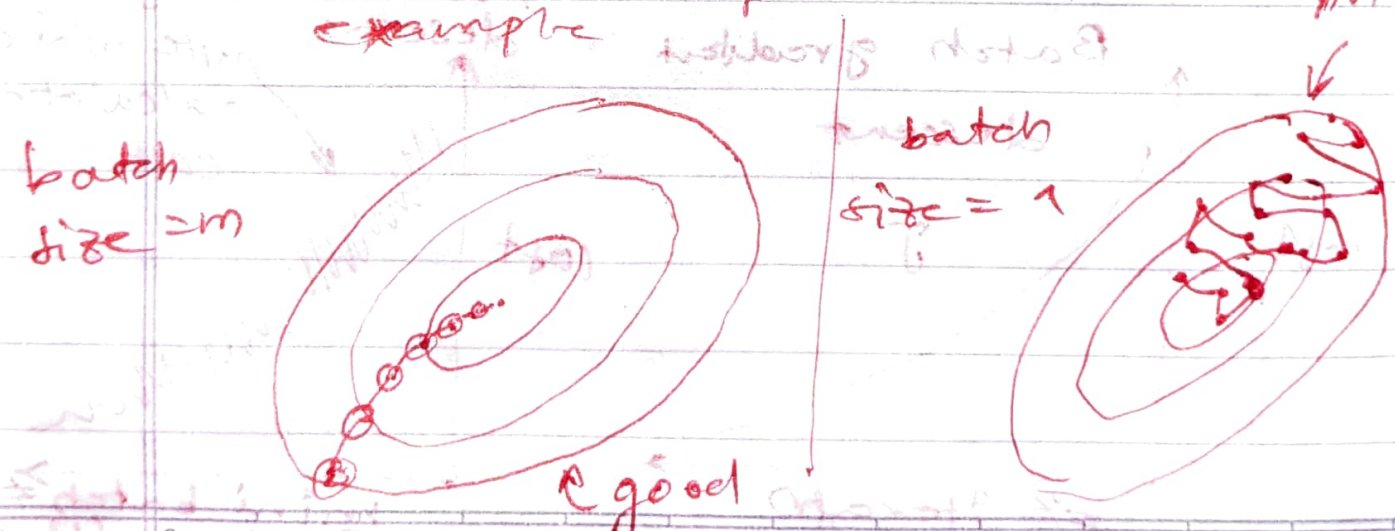
Mini batch size

if batch size = m (As good as OR same as batch gd.)

if batch size = 1 & ~~Echo~~
↳ (stochastic gd)

$(x^{\{t\}}, y^{\{t\}}) = [(x^{(1)}, y^{(1)}), (x^{2}, y^{(2)}), \ldots]$

Each training ~~example~~



Teacher's Signature

Using batch size = 1 is same as using for loops for training the model & not taking advantage of vectorization.

A mini batch size should be between 1 & m. [m = no. of training eg]

if $m \leq 2000$, don't do mini batch

Typical mini-batch size:

64, 128, 256, 512

2^6 2^7 2^8 2^9

power of 2's

Now mini-batch is also a hyperparameter

Exponentially weighted avgs.

formula

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

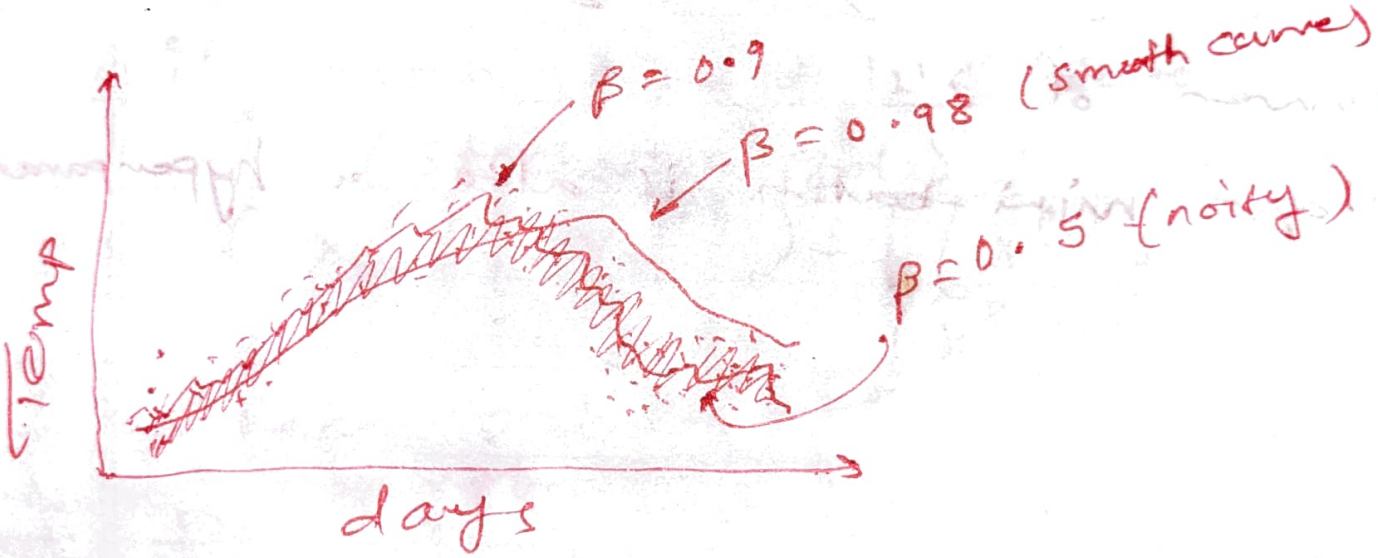
V_t of t^{th} time /example

V_t as approximately averaging over $\approx \frac{1}{1-\beta}$ days

if $\beta = 0.9$, V_t is avg. over 10 days

$\beta = 0.98$, ——— 50 days

$\beta = 0.5$, ——— 2 days



β is also a hyperparameter

Gradient descent with momentum
to speed up process of gd.

On iteration t:

Compute dW, db on the current mini-batch

momentum ↗ ↗ acceleration

$$V_{dW} = \beta V_{dW} + (1-\beta) dW$$
$$V_{db} = \beta V_{db} + (1-\beta) db$$
$$W = W - \alpha V_{dW}, \quad b = b - \alpha V_{db}$$

Analogy : ball rolling downhill

Hyperparameters : α, β

RMS prop [Root mean squared prop]

On iteration t:

Compute dW, db on curr. mini-batch

$$S_{dW} = \beta S_{dW} + (1-\beta) dW^2 \leftarrow \text{small}$$
↑ element wise
$$S_{db} = \beta S_{db} + (1-\beta) db^2 \leftarrow \text{large}$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}, \quad b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Adam optimizat^n algorithm

$V_{dw} = 0$, $S_{dw} = 0$ $V_{db} = 0$, $S_{db} = 0$

In Adam, we'll be ~~using~~ doing

bias correct^n

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$, $V_{db}^{com.} = V_{db}/(1-\beta_1^t)$

$S_{dw}^{corr} = S_{dw} / (1-\beta_2^t)$, $S_{db}^{corr} = S_{db}/(1-\beta_2^t)$

$w := w - \alpha \cdot \frac{V_{dw}^{com.}}{\sqrt{S_{dw}^{com.}} + \epsilon}$, $b := b - \alpha \frac{V_{db}^{com.}}{\sqrt{S_{db}^{com.}} + \epsilon}$

Why $\sqrt{S} + \epsilon$ in denom. ?

→ if $\sqrt{S} \approx 0$, the $\frac{V}{\sqrt{S}}$ term will be very large to prevent this ϵ is added

$\epsilon \approx 10^{-8}$

α : needs to tuned

β_1 : 0.9

β_2 : 0.999

ϵ : 10^{-8}

} Hyper-parameters

PAGE NO. :

DATE : / /

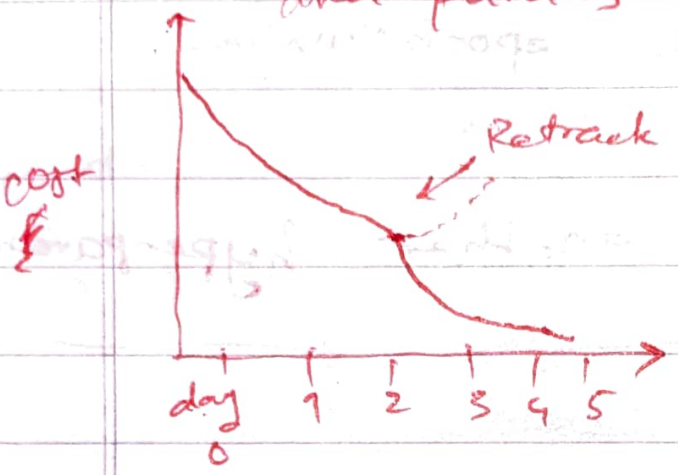
Learning rate decay (α)

$$\alpha = \frac{1}{1 + \text{decayRate} \times \text{epochNumber}} \alpha_0$$

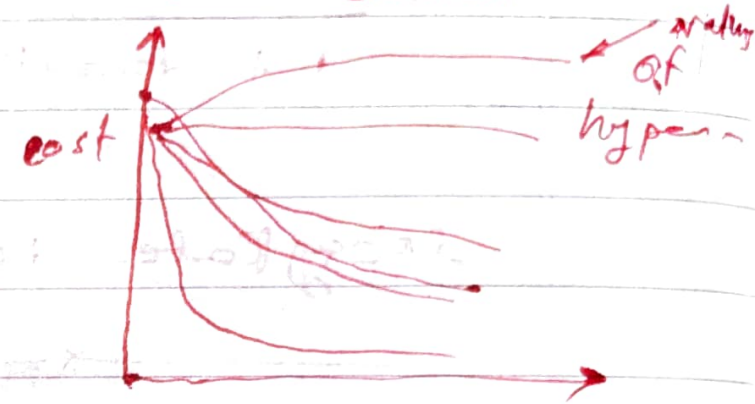
decayRate : another hyperparameter

Training models by changing hyperparameters

aka pandas



aka Caviar diff. starting of hyper-



Training the model by changing the values of hyper-parameters day-by-day to get best results

Time consuming but save lot of computational resources

Train many models in parallel with each having diff values of hyper parameters

faster but takes heavy toll of computational resources

Teacher's Signature

Normalising activations in a network

This makes training process smoother & efficient

Previously we used to perform normalization on input layer / features. Now, if we perform it on all layers, we would get better results. This is known as 'Batch normalization'.

Mean : $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$

$$X = X - \mu$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} \left[x^{(i)} - \mu\right]^2 \leftarrow \text{element-wise}$$

for input layer $\rightarrow$ $X = X / \sigma^2$

For l layers : $z^{(1)}, \ldots, z^{(m)}$ (for some l^{th} layer)

$z^{[l](i)}$

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i \left(z^{(i)} - \mu\right)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

Use $\tilde{z}^{(i)}$ instead of $z^{(i)}$

γ & β are learnable parameters

Batch norm as regularization

- Batch norm is applied on each mini batch. Mini-batch has size typically ranging from 64 to 512 (very small compared to all the training examples).
- Each mini batch is scaled by the mean / variance computed on just that mini-batch.
- This adds noise to the values $z^{[l]}$ within that minibatch. Similar to dropout, it adds noise to each hidden layer's activations.
- This has a slight regularization effect.

Dropout property:

minibatch : 64 ⟶ 512

more regularization — less regularization

Softmax (for C classes as output)

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{i=1}^{4} t_i}$$

say $z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$

$$a^{[l]} = g^{[l]}(z^{[l]}) = \begin{bmatrix} e^5/e^5+e^2+e^{-1}+e^3 \\ e^2/\text{---} \\ e^{-1}/\text{---} \\ e^3/\text{---} \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \leftarrow \text{hard max} \qquad = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix} \leftarrow \text{soft max}$$

Then loss function becomes :

$$L(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j$$

$$\text{cost} = J(w^{[1]}, b^{[1]}, \dots) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$$

$$Y = [y^{[1]}, y^{[2]}, \dots, y^{[m]}]$$

$$= \begin{bmatrix} 0 & 0 & & 0 \\ 1 & 0 & \cdots & 1 \\ 0 & 1 & & 0 \end{bmatrix}$$

Dimensions : $4 \times m$

Teacher's Signature